CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

LAB MANUAL

JULY 2025 – NOV 2025 / ODD SEMESTER

SUBJECT CODE/NAME: CS4303-DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY

YEAR/SEM: II/III

CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

Vision of the Institute

To be an eminent centre for Academia, Industry and Research by imparting knowledge, relevant practices and inculcating human values to address global challenges through novelty and sustainability

Mission of the Institute

IM1. To create next generation leaders by effective teaching learning methodologies and instill scientific spark in them to meet the global challenges.

IM2. To transform lives through deployment of emerging technology, novelty and sustainability.

IM3. To inculcate human values and ethical principles to cater to the societal needs.

IM4. To contribute towards the research ecosystem by providing a suitable, effective platform for interaction between industry, academia and R & D establishments.

IM5. To nurture incubation centres enabling structured entrepreneurship and start-ups.

Vision of the Department

To Excel in the emerging areas of Artificial Intelligence and Data Science by imparting knowledge, relevant practices and inculcating human values to transform the students as potential resources to contribute innovatively through advanced computing in real time situations.

Mission of the Department

M1: To provide strong fundamentals and technical skills for Computer Science applications through effective teaching learning methodologies.

M2: To transform lives of the students by nurturing ethical values, creativity and novelty to become Entrepreneurs and establish start-ups.

M3: To habituate the students to focus on sustainable solutions to improve the quality of life and the welfare of the society.

M4: To enhance the fabric of research in computing through collaborative linkages with industry and academia.

M5: To inculcate learning of the emerging technologies to pursue higher studies leading to lifelong learning.

CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

Program Educational Objectives Statements

PEO1: Contribute to the industry as an Engineer through sound knowledge acquired in core engineering to develop new processes and implement the solutions for industrial problems.

PEO2: Establish an organization / industry as an Entrepreneur with professionalism, leadership quality, teamwork, and ethical values to meet the societal needs.

PEO3: Create a better future by pursuing higher education / research and develop the sustainable products / solutions to meet the demand.

PROGRAM OUTCOMES

Learners should be able to:

PO1: **Engineering Knowledge**: Apply the knowledge of sciences like Mathematics, Basic Science, Engineering fundamental and their specialization to solve the complex engineering problems.

PO2: **Problem analysis**: Identify and analyze the user needs in the selection, creation, and evaluation and implementation process.

PO3: **Design/Development of Solution**: Identify the solutions for basic and complex engineering problems considering the needs of the society.

PO4: **Conduct Investigation of Complex Problem**: Conduct experiments, analyzed at and provide information.

PO5: **Modern tool usage**: Learn techniques, engineering and IT tools with their limitations and apply suitably.

PO6**: Engineer and Society**: Understand the impact of engineering solutions in a global and societal context.

PO7: **Environment and Sustainability**: Use engineering practices with a view to attain sustainability.

PO8: **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities.

PO9: Individual and teamwork: Function individually and in multi-disciplinary teams.

PO10: **Communication**: Communicate effectively in a professional and social environment through technical presentations, interpersonal skills, technical writing and graphics.

PO11: **Project Management**: Understand the basic engineering practices and to assist in the implementation of an effective project plan.

PO12: **Life-long learning**: Engage in life-long learning for the betterment of the society in general and individual in particular.

COURSE OUTCOMES:

At the end of this course, the students will be able to:

CO1: Understand the fundamental principles of problem solving and analyse the efficiency of algorithms using various frameworks

CO2: Apply Greedy Technique to Solve real time problems

CO3: Implement and evaluate various Divide and Conquer and Dynamic Programming Techniques

CO4: Apply Backtracking techniques to solve real-world problems.

CO5: Evaluate and Solve problems using approximation algorithms and randomized algorithms.

.

SYLLABUS

COURSE OBJECTIVES:

1. To understand and apply the algorithm analysis techniques on searching and sorting algorithms
2. To understand the different Greedy Algorithms.
3. To understand different algorithm design techniques
4. To solve programming problems using state space tree
5. To understand the concepts behind NP Completeness, Approximation algorithms and randomize algorithms.

UNIT I INTRODUCTION 9

Problem Solving : Programs and Algorithms-Problem Solving Aspects-Problem Solving Techniques-Algorithm analysis: Time and space complexity - Asymptotic Notations and its properties Best case, Worst case and average case analysis– Recurrence relation: substitution method – String Matching and searching: Interpolation Search, Pattern search: The naïve string- matching algorithm - Rabin-Karp algorithm - Knuth-Morris-Pratt algorithm.

UNIT - II GREEDY TECHNIQUE 9

Minimum spanning tree, Kruskal's and Prim's algorithm- Shortest path: Bellman-Ford algorithm - Dijkstra's algorithm - Floyd-Marshall algorithm - Network flow : Flow networks - Ford-Fulkerson method – Maximum bipartite matching.

UNIT - III DIVIDE-AND-CONQUER AND DYNAMIC PROGRAMMING 9

Divide and Conquer methodology: Finding maximum and minimum - Merge sort - Quick sort Dynamic programming: Elements of dynamic programming — Matrix-chain multiplication - Multi stage graph — Optimal Binary Search Trees - activity-selection problem –- Optimal Merge pattern-0/1 Knap sack problem.

UNIT - IV STATE SPACE SEARCH AND BACKTRACKING 9

Backtracking: n-Queens problem - Hamiltonian Circuit Problem - Subset Sum Problem –Graph colouring problem Branch and Bound: Solving 15-Puzzle problem – Assignment problemKnapsack Problem-Travelling Salesman Problem.

UNIT - V NP-COMPLETE AND APPROXIMATION ALGORITHM 9

Tractable and intractable problems: Polynomial time algorithms–Venndiagram representationNP-algorithms-NP-hardness and NP-completeness–Bin Packing problem- Problem reduction: TSP–3-CNF problem. Approximation Algorithms: TSP-Randomized Algorithms: concept and application- primality testing-randomized quicksort-Finding Kth smallest number.

TOTAL PERIODS: 45

Assignment

Assignment 1: Algorithm Analysis and Time Complexity

Title: Asymptotic Analysis of Sorting Algorithms Objective: Compare the time complexity of two sorting algorithms - Insertion Sort and Heap Sort.

Steps:

1. Provide a brief overview of Insertion Sort and Heap Sort algorithms.
2. Analyze the time complexity of each algorithm in terms of Big O notation.
3. Discuss the best-case, worst-case, and average-case scenarios for both algorithms.
4. Compare the efficiency of Insertion Sort and Heap Sort based on their time complexities.
5. Discuss any trade-offs or advantages one algorithm may have over the other.

Assignment 2: String Matching Algorithms

Title: Comparative Analysis of String-Matching Algorithms Objective: Analyze and compare the string-matching algorithms - Naïve String Matching, Rabin-Karp, and Knuth-Morris-Pratt.

Steps:

1. Provide a brief overview of each string-matching algorithm.
2. Analyze the time complexity of Naïve String Matching, Rabin-Karp, and Knuth- Morris-Pratt.
3. Discuss the best-case, worst-case, and average-case scenarios for each algorithm.
4. Compare the strengths and weaknesses of the algorithms, considering factors like pattern length and text size. Provide examples or scenarios where each algorithm may be most suitable.

Assignment 3: Divide and Conquer and Dynamic Programming

Title: Comparative Study of Divide and Conquer vs. Dynamic Programming Objective: Analyze and compare the efficiency of Divide and Conquer and Dynamic Programming approaches in solving two different problems - Finding Maximum and Minimum, and Matrix-Chain Multiplication.

Steps:

1. **Finding Maximum and Minimum:**
   - Explain the Divide and Conquer approach for finding the maximum and minimum in an array.
   - Analyze the time complexity of the Divide and Conquer solution.
   - Discuss any limitations or scenarios where this approach might be preferable.
2. **Matrix-Chain Multiplication:**
   - Describe the Dynamic Programming approach for solving the Matrix-Chain Multiplication problem.
   - Analyze the time and space complexity of the Dynamic Programming solution.
   - Discuss the advantages of using Dynamic Programming in this context.
3. **Comparison:**
   - Compare the Divide and Conquer and Dynamic Programming approaches in terms of time and space complexity.
   - Discuss scenarios where one approach might be more suitable than the other

Assignment 4: Greedy Technique and Dynamic Programming

Title: Greedy vs. Dynamic Programming in Optimization Problems Objective: Analyze and compare the Greedy Technique and Dynamic Programming approaches in solving two different optimization problems - Activity selection and Optimal Merge Pattern.

Steps:

1. **Activity-Selection Problem:**

- Explain the Greedy Strategy for solving the Activity-Selection problem.
- Analyze the time complexity of the Greedy solution.
- Discuss any potential pitfalls or limitations of the Greedy approach.

2. **Optimal Merge Pattern:**

- Describe the Dynamic Programming approach for solving the Optimal Merge Pattern problem.
- Analyze the time and space complexity of the Dynamic Programming solution.
- Discuss the advantages of using Dynamic Programming in this context.

3. **Comparison:**

- Compare the Greedy Technique and Dynamic Programmin
- approaches in terms of their suitability for optimization problems.
- Discuss scenarios where one approach might outperform the other or where a hybrid approach could be considered.

TOTAL: 75 PERIODS

COURSE OUTCOMES:

At the end of this course, the students will be able to:

CO1: Understand the fundamental principles of problem solving and analyse the efficiency of algorithms using various frameworks

CO2: Apply Greedy Technique to Solve real time problems

CO3: Implement and evaluate various Divide and Conquer and Dynamic Programming Techniques

CO4: Apply Backtracking techniques to solve real-world problems.

CO5: Evaluate and Solve problems using approximation algorithms and randomized algorithms

TEXTBOOKS:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", 3rd Edition, Prentice Hall of India, 2009.

2. Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran “Computer Algorithms/C++” Orient Blackswan, 2nd Edition, 2019.

PRACTICAL EXERCISES:

A. Searching and Sorting Algorithms

1. Implement Linear Search. Determine the time required to search for an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.

2. Implement recursive Binary Search. Determine the time required to search an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.

3. Given a text txt [0...n-1] and a pattern pat [0...m-1], write a function search (char pat [ ], char txt [ ]) that prints all occurrences of pat [ ] in txt[ ]. You may assume that n>m.

4. Sort a given set of elements using the Insertion Sort and Heap sort methods and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

B. Graph Algorithms

1. Develop a program to implement graph traversal using Breadth First Search.

2. Develop a program to implement graph traversal using Depth First Search.

3. From a given vertex in a weighted connected graph, develop a program to find the shortest paths to other vertices using Dijkstra's algorithm.

4. Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.

5. Implement Floyd's algorithm for the ALL-Pair Shortest path problem

6.. Compute the transitive closure of a given directed graph using Marshall's algorithm.

C. Algorithm Design Techniques

1. Develop a program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique.

2. Implement Merge sort and Quick sort methods to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

D. State Space Search Algorithms

1. Implement N Queens problem using Backtracking.

E. Approximation Algorithms Randomized Algorithms

1. Implement any scheme to find the optimal solution for the Traveling Sales person problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation.

2. Implement randomized algorithms for finding the kth smallest number. The programs can be implemented in C/C++/JAVA/Python.

EXP.NO:1	IMPLEMENTATION OF LINEAR SEARCH

AIM:

ALGORITHM:

PROGRAM:

```
import time
import matplotlib.pyplot as plt
import random
def linear_search(arr, x):
for i in range(len(arr)):
if arr[i] == x:
      return ireturn -1
n_values = [100, 1000, 10000, 100000, 1000000]
time_values = []
for n in n_values:

()linear_search(arr, x) end_time = time.time()
time_values.append(end_time - start_time)

plt.plot(n_values, time_values)plt.title('Linear Search')

plt.xlabel('Number of Elements')

plt.ylabel('Time Taken (seconds)')

plt.show()
```

OUTPUT:

Result:

EXP.NO:2	IMPLEMENTATION OF RECURSIVE BINARY SEARCH

AIM:

ALGORITHM:

PROGRAM:

```
import random
import time
import matplotlib.pyplot as plt
def binary_search_recursive(arr, low, high, x):if high >= low:mid = (high + low) // 2
      if arr[mid] == x:return mid
      elif arr[mid] > x:
 return binary_search_recursive(arr, low, mid - 1, x)
 else:
return binary_search_recursive(arr, mid + 1, high, x)
 else:
return -1
        def test_binary_search_recursive():
        sizes = [10, 100, 1000, 10000, 100000]
        times = []

for n in sizes:

arr = [random.randint(1, n) for _ in range(n)]
arr.sort()

start_time = time.time()x
= random.randint(1, n)
result = binary_search_recursive(arr, 0, n - 1, x)
end_time = time.time()

if result == -1:
```

```
    print(f"Element {x} not found in the array")
else:
    print(f"Element {x} found at index {result}")

elapsed_time = end_time - start_time

print(f"Time taken to search in array of size {n}: {elapsed_time:.6f} seconds")
times.append(elapsed_time)
print("="*50) # Plotting
plt.plot(sizes, times, marker='o')

plt.title("Recursive Binary Search Performance")
plt.xlabel("Size of Array")
plt.ylabel("Time Taken (in seconds)")
plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.show()
test_binary_search_recursive()
```

OUTPUT:

RESULT:

Ex.No :3	INTERPOLATION SEARCH

AIM:

ALGORITHM:

PROGRAM:

```
def interpolation_search(arr, x):low = 0
high = len(arr) - 1


while low <= high and arr[low] <= x <= arr[high]:
 # To prevent division by zero
if arr[low] == arr[high]:
if arr[low] == x:
        return low
    return -1

# Estimate the position of the target element
pos = low + ((high - low) // (arr[high] - arr[low]) * (x - arr[low]))


# Check if the estimated position holds the target elementif arr[pos] == x:
    return pos
```

```
        # If the target is larger, it is in the upper partif arr[pos] < x:
            low = pos + 1
        # If the target is smaller, it is in the lower partelse:
            high = pos - 1


    return -1  # Element is not in the array


# Example usage
arr = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
x = 70
result = interpolation_search(arr, x)if result
!= -1:
    print(f"Element found at index {result}")else:
    print("Element not found")
```

OUTPUT:

RESULT:

EX.NO:4	3 PATTERN MATCHING

AIM:

ALGORITHM:

PROGRAM:

```
def search(pat, txt):n = len(txt)
 m = len(pat)result = []

 for i in range(n - m + 1):j = 0
 while j < m:

 if txt[i + j] != pat[j]:
 break
 j += 1

 if j == m:
 result.append(i)
      return result

txt = "AABAACAADAABAABA"
pat = "AABA"
result = search(pat, txt)
print("Pattern found at indices:", result)
```

OUTPUT:

Result:

EXP.NO:5	IMPLEMENTATION OF INSERTION SORT

AIM:

ALGORITHM FOR INSERTION SORT:

PROGRAM:

```
import matplotlib.pyplot as plt
import  random
import time

def insertionSort(arr):n = len(arr)
   for i in range(1, n):
      key = arr[i]
      j = i - 1

      while j >= 0 and arr[j] > key:arr[j + 1] = arr[j]
         j -= 1

      arr[j + 1] = key


   def generateList(n):

   return [random.randint(1, 1000) for _ in range(n)]


   def measureTime(n): arr
   = generateList(n)
   startTime = time.time()
   insertionSort(arr) endTime
   = time.time()
   return endTime - startTime
```

```
    def plotGraph(nList):

    timeList = [measureTime(n) for n in nList]
    plt.plot(nList, timeList, 'o-')
    plt.xlabel('Number of Elements (n)') plt.ylabel('Time
    Taken (seconds)') plt.title('Insertion Sort Time
    Complexity')plt.show()

nList = [100, 500, 1000, 2000, 5000, 10000]

plotGraph(nList)
```

OUTPUT:

RESULT:

EXP.NO:6	IMPLEMENTATION OF HEAP SORT

AIM:

ALGORITHM FOR HEAP SORT:

PROGRAM:

```
import matplotlib.pyplot as plt
import random
import time

# Heapify a subtree rooted with node i
def heapify(arr, n, i):
largest = i  # Initialize largest as root l =
2 * i + 1  # left child
r = 2 * i + 2  # right child

# See if left child of root exists and is greater than rootif l < n and arr[i] < arr[l]:
    largest = l

# See if right child of root exists and is greater than rootif r < n and arr[largest] < arr[r]:
    largest = r

# Change root, if neededif
largest != i:
    arr[i], arr[largest] = arr[largest], arr[i]  # swap

    # Heapify the root
    heapify(arr, n, largest)

   # Heap sort function
   def heapSort(arr):
   n = len(arr)
```

```
    # Build a max heap

    for i in range(n // 2 - 1, -1, -1):
     heapify(arr, n, i)

     # Extract elements one by onefor i
      in    range(n - 1, 0, -1):
     arr[i], arr[0] = arr[0], arr[i] # swap
     heapify(arr, i, 0)

    # Generate a list of n random numbersdef generateList(n):
    return [random.randint(1, 1000) for i in range(n)]

    # Measure the time required to sort a list of n elementsdef measureTime(n):

        arr = generateList(n)
        startTime = time.time()
        heapSort(arr) endTime = time.time()
        return endTime - startTime

    # Plot a graph of the time required to sort a list of n elementsdef   plotGraph(nList):
  timeList = [measureTime(n) for n in nList]
  plt.plot(nList, timeList, 'o-')
  plt.xlabel('n')

  plt.ylabel('Time (s)')
  plt.title('HeapSort')
  plt.show()

nList = [100, 500, 1000, 2000, 5000, 10000]

plotGraph(nList)
```

OUTPUT:

RESULT:

EXP. NO: 7	IMPLEMENTATION OF GRAPH TRAVERSAL USING BREADTH-FIRSTSEARCH

AIM:

ALGORITHM:

PROGRAM:

```
import networkx as nx
import matplotlib.pyplot as plt

graph = {

'5': ['3', '7'],

'3': ['2', '4'],

'7': ['8'],

'2': [],

'4': ['8'],

'8': []

}

G = nx.Graph(graph) nx.draw(G, with_labels=True)
plt.show()

visited = []  # List for visited    nodesqueue = []   # Initialize a queue

def bfs(visited, graph, node):
visited.append(node) queue.append(node)
```

```
    while queue:

    m = queue.pop(0)
    print(m, end=" ")

    for neighbour in graph[m]:

        if neighbour not in visited:
            visited.append(neighbour)
            queue.append(neighbour)

# Driver Code

print("Following is the Breadth-First Search:")
bfs(visited, graph, '5')
```

OUTPUT:

RESULT:

EXP.NO:8	IMPLEMENTATION OF GRAPH TRAVERSAL USING DEPTH FIRST SEARCH

AIM:

ALGORITHM:

PROGRAM:

```
import networkx as nx
import matplotlib.pyplot as plt

# Using adjacency listg = {

   '5': ['3', '7'],

   '3': ['2', '4'],

   '7': ['8'],

   '2': [],

   '4': ['8'],

   '8': []
}
# Create a graph from the adjacency listG =
nx.Graph(g)
nx.draw(G, with_labels=True)
plt.show()

      # Set to keep track of visited nodesvisited = set()
         def dfs(visited, g,
         node):if node not in
         visited:
       print(node)
       visited.add(node)
       for neighbour in g[node]:
          dfs(visited, g, neighbour)

        # Driver code

        print("Following is the Depth-First Search:")dfs(visited, g, '5')
```

OUTPUT:

RESULT:

EXP.NO:9	IMPLEMENTATION OF DIJKSTRA'S ALGORITHM

AIM

ALGORITHM:

PROGRAM:

```
import networkx as nx
import matplotlib.pyplot as plt

# Create an empty undirected weighted graphG = nx.Graph()

# Define nodes

nodes_list = [1, 2, 3, 4, 5, 6, 7]
G.add_nodes_from(nodes_list)

# Add weighted edges
edges_list = [
   (1, 2, 13), (1, 4, 4), (2, 3, 2), (2, 4, 6), (2, 5, 4),

   (3, 5, 5), (3, 6, 6), (4, 5, 3), (4, 7, 4), (5, 6, 8),

   (5, 7, 7), (6, 7, 3)

]

G.add_weighted_edges_from(edges_list)


# Draw the graph
plt.figure()
pos = nx.spring_layout(G)
```

```
weight_labels = nx.get_edge_attributes(G, 'weight')

nx.draw(G, pos, font_color='white', node_shape='s', with_labels=True)
nx.draw_networkx_edge_labels(G, pos, edge_labels=weight_labels)

# Find shortest paths
p1 = nx.shortest_path(G, source=1, weight="weight")
# Shortest paths from node 1 to all nodes



p1to6 = nx.shortest_path(G, source=1, target=6, weight="weight")
# Shortest path from node1 to node 6


length = nx.shortest_path_length(G, source=1, target=6, weight="weight")
# Length of the shortest path from node 1 to node 6


# Print results

print("All shortest paths from node 1:", p1) print("Shortest
path from node 1 to node 6:", p1to6)
print("Length of the shortest path from node 1 to node 6:", length)


plt.show()
```

OUTPUT:

RESULT:

EXP.NO:10	IMPLEMENTATION OF PRIM'S ALGORITHM

AIM:

ALGORITHM:

PROGRAM:

```
import matplotlib.pyplot as plt
import networkx as nx
import pylab


# Create an empty undirected weighted graphG = nx.Graph()

# Add nodes

nodes_list = [1, 2, 3, 4, 5, 6, 7]
G.add_nodes_from(nodes_list)


# Add weighted edges
edges_list = [
(1, 2, 1), (1, 4, 4), (2, 3, 2), (2, 4, 6), (2, 5, 4),

(3, 5, 5), (3, 6, 6), (4, 5, 3), (4, 7, 4), (5, 6, 8),

(5, 7, 7)

]

G.add_weighted_edges_from(edges_list)

# Draw the graph

pos = nx.spring_layout(G)pylab.figure(1)
nx.draw(G, pos, with_labels=True)
# Use default edge labels nx.draw_networkx_edge_labels(G, pos, edge_labels=nx.get_edge_attributes(G,
'weight'))
```

```
# Calculate a minimum spanning tree of an undirected weighted graph using Prim'salgorithm
mst = nx.minimum_spanning_tree(G, algorithm='prim')

print("Edges in the minimum spanning tree:", sorted(mst.edges(data=True)))

plt.show()
```

OUTPUT:

RESULT:

EXP.NO:11	IMPLEMENTATION OF FLOYD'S ALGORITHM FOR THE ALL-PAIRS-SHORTEST-PATHS PROBLEM

AIM

ALGORITHM:

PROGRAM:

```
INF = float('inf')

def floyd_algorithm(graph):n = len(graph)
dist = [[INF for _ in range(n)] for _ in range(n)]

# Initialize the distance matrixfor i
in range(n):
  for j in range(n):

    if graph[i][j] != 0:

      dist[i][j] = graph[i][j]


# Floyd's algorithm to find all-pairs shortest pathsfor k in
range(n):
  for i in range(n): for
    j in range(n):
      if dist[i][k] + dist[k][j] < dist[i][j]:

        dist[i][j] = dist[i][k] + dist[k][j]
return dist
```

```
# Sample input graph
graph = [
    [0, 5, INF, 10],

    [INF, 0, 3, INF],

    [INF, INF, 0, 1],

    [INF, INF, INF, 0]

]


# Run the algorithm and print the resultresult =
floyd_algorithm(graph)
for row in result:
    print(row)
```

OUTPUT:

RESULT:

EXP.NO:12	COMPUTATION OF THE TRANSITIVE CLOSURE OF A DIRECTED GRAPHUSING WARSHALL'S ALGORITHM

AIM:

ALGORITHM:

PROGRAM:

```
def warshall_algorithm(graph):n =
   len(graph)

   # Create a copy of the original graph
   transitive_closure = [row[:] for row in graph]

   # Compute the transitive closure using Warshall's algorithmfor k in
   range(n):
      for i in range(n): for
         j in range(n):
            transitive_closure[i][j] = (transitive_closure[i][j] or (transitive_closure[i][k] and
                                 transitive_closure[k][j]))

   return transitive_closure
```

```
# Sample input
graph = [
    [0, 1, 0, 0],

    [0, 0, 1, 0],

    [0, 0, 0, 1],

    [1, 0, 0, 0]

]

# Run the algorithm and print the resultresult =
warshall_algorithm(graph)
for row in result:
    print(row)
```

OUTPUT:

RESULT:

EXP.NO:13	IMPLEMENTATION OF FINDING THE MAXIMUM AND MINIMUM NUMBERSIN A LIST USING DIVIDE AND CONQUER TECHNIQUE

AIM

ALGORITHM:

PROGRAM:

```
def find_max_min(arr):
if len(arr) == 1:
return arr[0], arr[0]elif
len(arr) == 2:
if arr[0] > arr[1]: return
arr[0], arr[1]
else:

return arr[1], arr[0]
else:
mid = len(arr) // 2
left_max, left_min = find_max_min(arr[:mid]) right_max,
right_min = find_max_min(arr[mid:])
return max(left_max, right_max), min(left_min, right_min)

# Example usage arr =
[3, 1, 5, 2, 9, 7]
max_num, min_num = find_max_min(arr)
print("Maximum number:", max_num)
print("Minimum number:", min_num)
```

OUTPUT:

RESULT:

EXP.NO:14	IMPLEMENTATION OF MERGE SORT

AIM:

ALGORITHM:

PROGRAM:

```
import random
import time
import matplotlib.pyplot as plt


def merge_sort(arr):if
   len(arr) > 1:
      mid = len(arr) // 2
      left_half = arr[:mid]
      right_half = arr[mid:]

      merge_sort(left_half)
      merge_sort(right_half)

      i = j = k = 0

      while i < len(left_half) and j < len(right_half):if
         left_half[i] < right_half[j]:
            arr[k] = left_half[i] i
            += 1
         else:

            arr[k] = right_half[j] j
            += 1
         k += 1
```

```
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1

            k += 1


        while j < len(right_half):arr[k] =
            right_half[j]
            j += 1

            k += 1


def test_merge_sort(n):

    arr = [random.randint(1, 100) for _ in range(n)]start_time =
    time.time()
    merge_sort(arr) end_time =
    time.time()
    return end_time - start_time


if__name___== '_main__':

    ns = [10, 100, 1000, 10000, 100000]

    times = []
    for n in ns:

        t = test_merge_sort(n)
        times.append(t)
        print(f"Merge sort took {t:.6f} seconds to sort {n} elements.")


    plt.plot(ns, times, 'o-') plt.xlabel('Number of
    elements (n)') plt.ylabel('Time taken (s)')
    plt.title('Merge Sort Time Complexity')
    plt.show()
```

OUTPUT:

RESULT:

EXP.NO:15	IMPLEMENTATION OF QUICK SORT

AIM:

` **ALGORITHM:**

PROGRAM:

```
import randomimport time
import matplotlib.pyplot as plt

def quicksort(arr):if len(arr) <= 1: return arr
pivot = arr[0]left = []
right = []

  for i in range(1, len(arr)):if arr[i] < pivot:
  left.append(arr[i])else:
  right.append(arr[i])

   return quicksort(left) + [pivot] + quicksort(right)


  def measure_time(n, num_repeats=5):times = []
   for _ in range(num_repeats):

arr = [random.randint(0, 1000000) for _ in range(n)]start_time = time.time()
```

```
    quicksort(arr) end_time = time.time()
    times.append(end_time - start_time)return sum(times) / len(times)

  if__name_== '_main_':num_repeats = 10
max_n = 10000

step_size = 1000

ns = range(1000, max_n + step_size, step_size)times = []

for n in ns:

    t = measure_time(n, num_repeats)
    times.append(t)
    print(f"Quick Sort took {t:.6f} seconds to sort {n} elements.")


plt.figure(figsize=(10, 6))

plt.plot(ns, times, 'o-', color='green')

plt.yscale('log') # Use logarithmic scale for better visualization
plt.xscale('log') # Use logarithmic scale for better visualization
plt.xlabel('Number of elements (n)')
plt.ylabel('Time taken (s)') plt.title('Quick
Sort Time Complexity')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

OUTPUT:

RESULT:

EXP.NO:16	IMPLEMENTATION OF N-QUEENS PROBLEM USING BACKTRACKING

AIM

ALGORITHM:

PROGRAM:

```
def is_safe(board, row, col, n):

    # Check if there is any queen in the same row
    for i in range(col):

        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):if
        board[i][j] == 1:
            return False
```

```
    # Check lower diagonal on left side

    for i, j in zip(range(row, n, 1), range(col, -1, -1)):if
        board[i][j] == 1:
            return False


    return True


def solve_n_queens(board, col, n):if col
    >= n:
        # All queens have been placed successfullyreturn True

    for row in range(n):

        if is_safe(board, row, col, n):

            # Place the queen in the current cell
            board[row][col] = 1
            # Recur to place the rest of the queensif
            solve_n_queens(board, col + 1, n):
                return True


            # Backtrack and remove the queen from the current cellboard[row][col] = 0

return False


def print_board(board, n):for i
    in range(n):
        for j in range(n):

            print("Q" if board[i][j] else ".", end=" ")print()

def n_queens(n):

    # Initialize the board

    board = [[0 for _ in range(n)] for _ in range(n)]


    if not solve_n_queens(board, 0, n):
        print("Solution does not exist.") return
        False

    print("Solution:")
    print_board(board, n)
    return True

if__name____== "_main_":

    n = int(input("Enter the number of queens: "))n_queens(n)
```

OUTPUT:

RESULT:

EX.NO:17	IMPLEMENTATION OF ANY SCHEME TO FIND THE OPTIMAL SOLUTION FOR THE TRAVELING SALES PERSON PROBLEM

AIM:

ALGORITHM:

PROGRAM:

```
import itertools
import time

def distance(city1, city2):

    # Assuming distance is Euclidean distance

    return ((city1[0] - city2[0]) ** 2 + (city1[1] - city2[1]) ** 2) ** 0.5


def tsp_brute_force(cities):

    # Calculate all possible permutations of the cities
    permutations = itertools.permutations(cities)
    # Initialize the shortest path to infinity
    shortest_path = float('inf') shortest_path_order
    = None

    # Iterate over all permutations to find the shortest pathfor
    permutation in permutations:
        path_length = 0
```

```
        for i in range(len(permutation) - 1):

            path_length += distance(permutation[i], permutation[i + 1])path_length +=
        distance(permutation[-1], permutation[0])

        # Update the shortest path if the current path is shorterif
        path_length < shortest_path:
            shortest_path = path_length
            shortest_path_order = permutation

    return shortest_path, shortest_path_order


def tsp_nearest_neighbor(cities):

    # Start with the first city in the list as the current city
    current_city = cities[0]
    visited_cities = [current_city] unvisited_cities =
    set(cities[1:])

    # Iterate over all cities to find the nearest neighbor
    while unvisited_cities:
        nearest_neighbor = None
        nearest_distance = float('inf')

        for city in unvisited_cities:

            distance_to_city = distance(current_city, city)if
            distance_to_city < nearest_distance:
                nearest_distance = distance_to_city
                nearest_neighbor = city

        visited_cities.append(nearest_neighbor)
        unvisited_cities.remove(nearest_neighbor)
        current_city = nearest_neighbor

    # Calculate the total distance of the path
    total_distance = 0
    for i in range(len(visited_cities) - 1):

        total_distance += distance(visited_cities[i], visited_cities[i + 1])total_distance +=
    distance(visited_cities[-1], visited_cities[0])

    return total_distance, visited_cities


# Generate a list of random cities

cities = [(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9)]
```

```
# Find the optimal solution using brute forcestart_time
= time.time()
optimal_path_length, optimal_path_order = tsp_brute_force(cities)end_time =
time.time()
print("Optimal path length:", optimal_path_length)print("Optimal
path order:", optimal_path_order)
print("Time taken (brute force):", end_time - start_time, "seconds")


# Find the approximate solution using the nearest neighbor algorithmstart_time =
time.time()
approximate_path_length, approximate_path_order = tsp_nearest_neighbor(cities) end_time = time.time()
print("Approximate path length:", approximate_path_length)print("Approximate path
order:", approximate_path_order)
print("Time taken (nearest neighbor):", end_time - start_time, "seconds")
```

OUTPUT:

RESULT:

EX.NO:18	IMPLEMENTATIONOFRANDOMIZEDALGORITHMSFORFINDINGTHEKTHSMALLESTNUMBER

AIM

ALGORITHM:

PROGRAM:

```
import random

# Function to partition the array around a pivotdef partition(arr, low, high):
    pivot = arr[high] i
    = low - 1
    for j in range(low, high):if
        arr[j] <= pivot:
            i += 1

            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]return i
    + 1

# QuickSelect function to find the k-th smallest elementdef
quickselect(arr, low, high, k):
    if low < high:

        pivot_index = random.randint(low, high) arr[pivot_index],
        arr[high] = arr[high], arr[pivot_index]

        pivot_pos = partition(arr, low, high)if k ==
        pivot_pos:
            return  arr[k] elif
```

```
        k < pivot_pos:
        return quickselect(arr, low, pivot_pos - 1, k)else:
        return quickselect(arr, pivot_pos + 1, high, k)else:
    return arr[low]


# Function to find the k-th smallest element in the arraydef
find_kth_smallest(arr, k):
  if k < 1 or k > len(arr):

      raise ValueError("k is out of bounds")
  return quickselect(arr, 0, len(arr) - 1, k - 1)


# Example usage

arr = [7, 10, 4, 3, 20, 15]

k = 3

print(f"The {k}-th smallest element is {find_kth_smallest(arr, k)}")
```

OUTPUT:

RESULT